Docker - Material de apoio, Matheus Luis Oliveira da Silva

▼ Definição do projeto

1. Teórico (informativo):
    - Containers (o que são, história, docker, utilidade, prós vs contras)
    - “Runtimes” de containers
2. Prático:
    - Setup de ambiente na máquina local para as práticas (WSL2 se for Windows, Docker para Linux)
    - Docker (linha de comandos mais utilizadas: construção de imagem, execução de containers, uso de repositório, comunicação entre containers, volumes, rede)
    - docker-compose

- Entregáveis (serão duas entregas **individuais e em datas diferentes**, que deverão ser **combinadas** com os mentores):
    1. Apresentação dos conteúdos, com o intuito de demonstrar domínio sobre os assuntos
    2. Apresentação prática de uma implementação de solução (projetinho) mostrando a integração de containers com o uso do docker-compose. Exemplo sugerido: frontend + backend + bancos de dados - **de projetos já existentes** (ex: nginx + Wordpress + MySQL). O intuito é mostrar a subida da solução e a comunicação dos contêineres.
- Material de apoio:

- Site docker: https://www.docker.com
- Site alura (cursos): https://www.alura.com.br/ **(o mentor passará o acesso a cada um de vocês individualmente)**
- Docker Tutorial for Beginners: https://www.youtube.com/watch?v=zJ6WbK9zFpI
- Canal da Nana (youtube): https://www.youtube.com/c/TechWorldwithNana
- https://cursos.alura.com.br/course/docker-e-docker-compose

▼ **Apresentação Teórica**

▼ Refs:

https://4linux.com.br/o-que-e-docker/

https://www.redhat.com/pt-br/topics/containers/what-is-docker

https://solvimm.com/blog/o-que-e-docker/#:~:text=Histórico,seu fornecimento através da nuvem.

https://etcd.dev/2021/09/10/a-historia-dos-containers-linux/

https://www.youtube.com/watch?v=zJ6WbK9zFpI&t=329s

▼ História:

- Ferramenta para gerenciamento e criação de contêineres de aplicação;
- Criada em 2013 pela dotCloud, futura Docker;
- Motivada pela alta demanda por maquinas virtuais, as quais passaram a causar problemas por conta do alto custo de hardware necessário;

Desde o princípio:

- chroot 1979 → Alterar o diretório raiz de um processo (inicio do isolamento);
- FreeBSD Jails, LinuxVServer 2000 → Possibilidade de subdivisão do SO em segmentos menores com seu próprio hostname, diretório e endereço de rede. (início da virtualização “leve”);
- Solaris Containers Oracle 2004 → Inicio dos containers, particionando recursos e sistemas de arquivos e volumes;

- Process Containers Google 2006 → Limitar, contabilizar e isolar uma arvore de processos, control groups, cgroups;
- LXC 2008 → primeira e mais completa ferramenta de contêinerização, utilizando de namespaces e cgroups, num unico kernel Linux;
- Warden cloud Foundry 2011 → Isolava ambientes em qualquer SO;
- Docker 2013 → Também começou com o LXC, mas posteriormente criou sua própria biblioteca, explodiu em popularidade.

▼ Problemáticas:

- Compatibilidade e dependências:
  - Uma vasto conjunto de componentes para um único sistema que tem variados requisitos em relação ao sistema operacional e suas versões, bibliotecas, e demais dependências;
    - A princípio, pensar em compatibilidade apenas uma vez não parece um problema grave, entretanto, ao pensar em atualizações atômicas, em um componente ou serviço específico da aplicação, toda a compatibilidade entre a nova feature e o sistema atual deve ser verificada novamente;
- Tempo de configuração:
  - Com um grande número de diferentes serviços, dependências, bibliotecas e ambientes distintos trabalhando em conjunto, quando um desenvolvedor passa a trabalhar com o sistema em questão um longo tempo de adaptação e de preparação é demandado quando se tem de configurar cada componente manualmente, assim como interconectá-los;
- Diferentes ambientes de uso:
  - Além da interdependência entre as componentes de um sistema, deve-se levar em conta a dependência de todos esses com os ambientes que o usuário trabalha, pois ao considerar toda a gama possível de ambientes que o usuário pode preferir utilizar, seja um SO específico, uma ferramenta, interface, etc., a compatibilidade também passa a ser um problema;

▼ Utilidade / Solução:

- Uma interface que permita modificar os componentes sem que os outros componentes tenham de ser modificados ou o SO adaptado;
- Pode-se, então, rodar cada componente em um local separado, com suas próprias dependências e bibliotecas;
- Containers compartilham o mesmo Kernel do SO:
    - Ou seja, o docker se encontra acima do kernel, e gerencia os contêineres baseados em qualquer distribuição de software que seja compatível com aquele kernel: Ou seja, poderia ter um kernel Linux - ubuntu, e rodar contêineres baseados em fedora, arch Linux, CentOS, etc;

      Entretanto: Não é possível rodar um container baseado em windows no docker hospedado num kernel Linux. Por conta disso, precisamos da virtualização do Linux para utilizar o docker;
- Open Container Initiative
    - “The **Open Container Initiative** is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes.”
- *Runtimes* de Containers
    - Um breve resumo do processamento de uma execução de um container:
        - The image is pulled from an image registry if it not available locally

          The image is extracted onto a copy-on-write filesystem, and all the container layers overlay each other to create a merged filesystem

          A container mount point is prepared

          Metadata is set from the container image, including settings like overriding CMD, ENTRYPOINT from user inputs, setting up SECCOMP rules, etc., to ensure container runs as expected

          The kernel is alerted to assign some sort of isolation, such as process, networking, and filesystem, to this container (namespaces)

The kernel is also alerted to assign some resource limits like CPU or memory limits to this container (cgroups)

A system call (syscall) is passed to the kernel to start the container

- Para executar, precisamos de um serviço que gerencie todos esses procedimentos, são os chamados runtimes de container.
- Exemplos de Runtimes:
  - Docker
  - CRI-O
  - PodMan
  - Kata
  - PouchContainer

▼ Diferenças entre containers e VM’s

- Maquinas Virtuais:
  - Usam quantidades significativas de recursos, dada a necessidade de virtualizar todo o SO + kernel para cada VM em execução.
  - Tem um tamanho em disco considerável, dado que as VM’s são grandes e complexas.
  - Tempo de Inicialização alto, para subir todo o sistema operacional, suas dependências e softwares próprios.

- Contêineres:
  - Usam menos recursos, uma vez que que são mais leves, partindo do mesmo SO, virtualizam apenas o ambiente de desenvolvimento e processamento.
  - Usam menos espaço de disco, uma vez que são menores e possuem menos componentes.
  - Inicializam mais rápido, pelos mesmos motivos.
  - Entretanto, os contêineres necessitam do compartilhamento de recursos que não são necessários quando se utiliza uma VM, como o kernel, que no caso, possuem o seu próprio.

- Para entender a estrutura de dependências, é necessário que se compreenda cada um dos componentes envolvidos:
    - Kernel: Intermediário entre o sistema Operacional e o hardware da maquina hospedeira, responsável pelo gerenciamento e distribuição de recursos para os processos do SO
    - Hypervisor:
        - Tipo 1: São integrados diretamente ao hardware
        - Tipo 2: São intermediados pelo kernel da máquina hospedeira
        - Gerenciam a criação e processamento de outras VM's, multiplexando o hardware para cada um dos kernels virtualizados. Inclusive, possui, geralmente, maior privilégio que o kernel.
    - SO: Software que gerencia os processos das aplicações, conectando o kernel as aplicações finais.

▼ Prós e Contras

Prós

- Isolamento a nível de aplicação;
- Mais rápido para entrar em execução e usa menos memoria;
- Mais fácil de migrar, voltar a uma versão antiga e ser transportado;
- Sistema de versionamento de imagens de mais fácil acesso;
- Alinhado com arquitetura de micro serviços.

Contras

- Ainda muito novos quando comparados a VM 2013 vs 1960
- Poucos profissionais por enquanto
- Seria só mais um modismo?
- Problemas de segurança?:

        - "Docker is about running random code downloaded from the Internet and running it as root."
        - Não esquecer que container services são serviços como quaisquer outros
        - Privilégios de root devem ser evitados ou no mínimo tratados
        - O que levar em consideração:
            - Ao rodar um container, namespaces são criados, isolando o container de outros container e do docker host
            - Sua rede é totalmente personalizável, com as pontes que perfuram o isolamento ficando a cargo do desenvolvedor.
            - Control groups defendem o sistema contra DoS, gerenciando os recursos utilizados por cada container
            - Atenção ao privilégio do docker daemon (que deve ser rodado com root privilege)

▼ **Docker e seus componentes**

▼ Refs

https://solvimm.com/blog/o-que-e-docker/#:~:text=Histórico,seu fornecimento através da nuvem.

https://www.simplilearn.com/tutorials/docker-tutorial/what-is-docker

https://blog.knoldus.com/docker-components/

https://docs.docker.com/get-started/overview/

https://www.ctl.io/developers/blog/post/optimizing-docker-images/

- Containers:
    - Conjunto de um ou mais processos isolados do sistema;
    - Imagem: A imagem é o pacote que possui todos os arquivos, bibliotecas e dependências necessárias para rodar esses processos, torna-se um container quando está em execução;
- LXC X Docker:
    - A LXC permitia a execução de um sistema linux como um todo, isolado, entretanto o docker encorajava a segmentação da aplicação em

processos menores, lidando diretamente com eles, assim como oferecia features especificas para configuração individual desses serviços.

- Docker:
  - Escrito em Go Language, com profunda interação com o kernel linux;
  - Usa de namespaces para garantir o isolamento de um container com um determinado conjunto de recursos do kernel, que é diferente dos demais processos ou containers em execução;
  - Client-Server Architecture:
    - O cliente fala com o daemon, enviando comandos para configurar, rodar e distribuir os containers;
    - Ambos podem ou não estar no mesmo sistema;
    - Comunicação via REST API, através de Unix sockets ou rede;
      - Representational State Transfer;
      - Application programming interface;
      - Assim, o client envia uma requisição para o host via API, o qual hospeda o daemon que executará as requisições feitas, como build, pull, run, etc;
  - Docker Registry: Local que armazena as docker images, de onde comando como docker pull ou docker run puxam as imagens para serem executadas;
  - Images: `docker build -Dockerfile-`
    - É um template com instruções de como montar um container docker, podendo, inclusive, combinar instruções novas com imagens já existentes.;
    - Esta contida em um Dockerfile, o qual possui instruções, linha a linha, cada qual gera uma layer da imagem. Assim, cada alteração feita na imagem, altera somente a layer que foi modificada, mantendo as demais intactas.;
  - Layers:
    - Cada instrução numa imagem se chama layer;
    - Curiosamente, cada layer é, também, uma imagem (??);

        - Ou seja, uma imagem nada mais é do que um conjunto de imagens;
        - Uma layer adicionada somente aumenta o tamanho da imagem, dado que “o passado permanece”, ou seja, mesmo que eu altere alguma layer passada e já commitada, ambas permanecerão acessíveis via tag;
        - Para reduzir o tamanho da imagem, é possível concatenar comandos que são triviais, com cuidado pois os intermediarios não serão commitados;
        - Entre as camadas de uma imagem, imagens em comum são reutilizadas ? (cache);
    - Docker Cache:
        - Docker possui cache, ou seja, quando uma imagem é criada, ela é alocada numa cache local, se ela for reconstruída sem ter nenhum parent node alterado, ela será rapidamente recuperada da cache;
        - Inclusive, quando uma imagem é alterada, a imagem anterior permanece na cache, criando apenas mais um filho no parent node;
    - Containers: `docker run -Image-`
        - Instância de execução de uma imagem;
        - Segue todas as descrições contidas na imagem;
        - Isolamento variável, de acordo com as especificações;
        - Depende de um processo em execução para “viver”;
        - Quando removido, todas as alterações não armazenadas são perdidas.
    - Exemplo:
        - `docker run -i -t ubuntu /bin/bash`;
        - Docker importará a imagem caso não exista localmente: `docker pull ubuntu`;
        - Docker cria um novo container: `docker container create`;
        - Docker aloca um sistema de arquivos de leitura e escrita para que possa criar e modificar arquivos nesse sistema local;

        - Docker cria uma interface de rede par ao container que pode se conectar a redes externas via conexão com a interface do host do container;
        - O container executa /bin/bash e aguarda novos comandos;
    - Docker Storage:
        - Todo dado escrito num container é perdido ao seu desligamento se não armazenado numa unidade de armazenamento persistente.
        - Camada Read-write, e procedimento copy-on-write. Por conta disso, também, os containers são muito leves!!
        - Data volumes: instala uma unidade de armazenamento no filesystem do docker conectado ao host do container, só pode ser alterado por processos do docker. Compatível com hosts remotos ou serviços de nuvem.

          Instalado no diretório /var/lib/docker
        - Bind Mounts: pode instalar uma unidade em qualquer lugar do sistema de arquivos do host, mesmo que fora dos domínios do docker, pode ser alterado por qualquer processo.
        - TMPFS: volume temporário criado na memoria do host, usado para arquivos sensíveis que não devem ser gravados na camada de read-write
    - Docker network:
        - Bridge network:
            - Permite trafego entre dois segmentos de rede;
            - Uma docker bridge permite conexão entre containers de uma mesma bridge e num mesmo docker host, cada container recebe um IP que o identifica;
            - Impede a conexão com containers fora da bridge;
            - Para containers externos ao docker daemon host, usar overlay network;
            - Por padrão, existe uma default bridge, não recomendada: comunicação somente via Ip, conexão automatica de qualquer container criado, necessidade de links manuais;

        - User-Defined bridge:
            - Pode realizar comunicação via nome;
            - Pode conectar e desconectar containers em execução, sem necessidade de derrubá-los;
            - Melhor isolamento;
        - Host network:
            - O container passa a utilizar o IP do host em que está, o que não permite mais que mais de um app utilize a mesma porta.
        - Overlay network:
            - Permite comunicação entre containers em diferentes daemons hosts
            - Permite comunicação entre containers isolados e, inclusive, swarm service containers
            - Redes padrões `ingress` e `docker_gwbridge`, também customizáveis com user-defined
        - IPvlan e MACvlan:
            - Além de endereços lógicos de IP, os containers podem também ter endereços físicos virtualizados, correspondentes às suas interfaces de rede simuladas.
        - Nenhuma rede:
            - É possível desativar a rede de um container ao executá-lo;
            - Obviamente, não é possível para swarm services;
            - Apenas o loopback permanece
    - Docker Compose:
        - Roda múltiplos containers como um único serviço;
        - Permite a comunicação entre os containers com facilidade;
        - Basicamente encapsula, novamente, a montagem de vários containers num único arquivo de instruções;
    - Docker Swarm:

        - Também permite que múltiplos containers rodem simultaneamente, porém agora podendo ser esses de diferentes daemons;
        - os daemons por sua vez também se comunicam, via Docker API
        - Worker nodes executem as tarefas, enquanto os manager nodes delegam as tarefas e realizam as devidas comunicações

▼ **Implementação Prática**

▼ Comandos e Execução:

- Baixar uma imagem do dockerHub
    - `docker pull IMAGENAME`
- Logar no dockerHub, etiquetar imagem criada, fazer upload da imagem
    - `docker login -u USERNAME`
    - `docker tag USERNAME/IMAGETOCOPY:VERSION USERNAME/IMAGENAME:VERSION`
    - `docker push USERNAME/IMAGENAME:VERSION`
- Construir imagem a partir do Dockerfile, -t para dar o nome da imagem seguido do diretório para contexto de build
    - `docker build -t USERNAME/IMAGENAME:VERSION CONTEXTDIRECTORY`
- Listar containers, imagens, networks, volumes, mapeamento de portas
    - `docker ps -a`
    - `docker images`
    - `docker network ls`
    - `docker volume ls`
    - `docker port CONTAINERNAME`
- Instanciar uma imagem, parar um container, subir um container
    - `docker run IMAGENAME`
        - `docker run IMAGENAME COMMAND` (para iniciar uma imagem como um comando de entrada)
    - `docker stop CONTAINERNAME`
    - `docker start CONTAINERNAME`
- Remover um container, imagem, volume…

    - `docker rmi IMAGENAME`
    - `docker rm CONTAINERNAME`
    - `docker volume rm VOLUMENAME`
- Executa um comando em um container que **está em execução**
    - `docker exec CONTAINERNAMEORID COMMAND`
        - `docker exec -w DIRECTORY CONTAINERNAME COMMAND` (para especificar o diretório de execução)
- Conectar ou desconectar o terminal ao container em execução:
    - `docker attach CONTAINERNAME`
    - `docker run IMAGENAME -d` (modo de execução em background)
- Lista todas as informações a respeito do container
    - `docker inspect CONTAINERNAME`
- Lista as informações da árvore de montagem da imagem
    - `docker history IMAGEID`
- Escreve no terminal todo o log de execução de um dado container
    - `docker logs CONTAINERNAME`
- Cria uma network ou um volume
    - `docker network create \ –driver bridge \ –subnet 182.18.0.0:16 custom-isolated-network`
    - `docker volume create VOLUMENAME`
- Copia conteúdos do container para o host evice versa:
    - `docker cp ./some_file CONTAINER:/work`

▼ Run

- --name = CONTAINERNAME
- --network = none, host
- --cpus=.5
- --memory=100M
- -H=ENGINEADDRESS:PORT

- -d detach mode
- -i permite stdin
- -t permite um pseudo terminal
- -p mapeamento de porta
  - -p HOSTPORT:SERVICEPORT
- -v mapeamento de volume
  - -v VOLUMENAME:CONTAINERSTORELOCATION
  - --mount type=bind, source=HOSTDIRECTORY,target=CONTAINERDIRECTORY
  - --mount source=MyVolume, target=CONTAINERDIRECTORY
- -e variáveis de ambiente
  - -e NOMEVARIAVEL=VALOR

▼ DockerFile

- FROM → Imagem base da qual anova imagem será criada
- WORKDIR → Diretorio atual
- ARG → Variaveis de build da imagem
- ENV → Variaveis de ambiente para container
- EXPOSE → expor uma porta específica
- RUN → Comandos que serão executados no build da imagem
- COPY → Comando para copiar diretórios do sistema local para um diretorio dentro da imagem
- CMD → Comando que será executado quando a imagem for executada, substituído totalmente quando sobrescrito na chamada do docker
- ENTRYPOINT → Comando que será executado quando a imagem for executada, concatenado com a passagem de parametro na chamada do docker
- CMD + ENTRYPOINT → O CMD servirá como comportamento padrão

▼ Docker Compose

- Composição de todo o ambiente num só arquivo yml

- comando docker compose up para iniciar todos os serviços de uma só vez
- Estrutura básica com o mínimo para rodar uma aplicação minimamente completa:
- `services:`
  - `SERVICE1:`
    - `build:`
    - `containername:`
    - `image:`
    - `ports:`
    - `links or networks:`
    - `depends_on:`
- `networks:`
  - `-NETWORKNAME`
  - `-NETWORKNAME2`
- `volumes:`
  - `-VOLUMENAME`
  - `-VOLUMENAME2`